Shader Algebra

Michael McCool Stefanus Du Toit Tiberiu Popa Bryan Chan Kevin Moule

Computer Graphics Lab, School of Computer Science, University of Waterloo

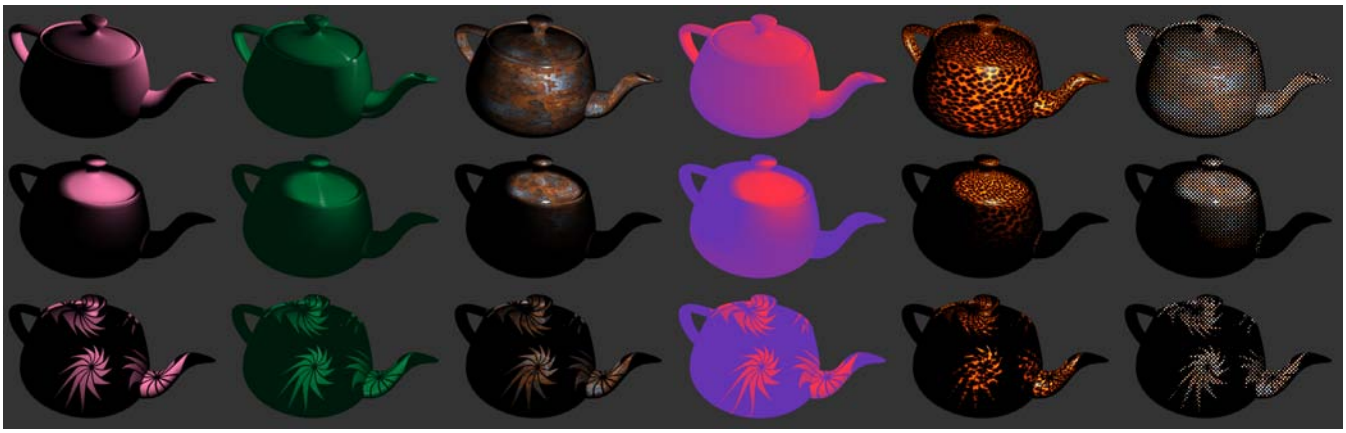

Figure 1: Some applications of shader algebra operations: surface and light shaders can be combined, output colors can be processed (in this case, halftoning was applied), and any attribute or parameter can be replaced with a texture map.

Abstract

An algebra consists of a set of objects and a set of operators that act on those objects. We treat shader programs as first-class objects and define two operators: connection and combination. Connection is functional composition: the outputs of one shader are fed into the inputs of another. Combination concatenates the input channels, output channels, and computations of two shaders. Similar operators can be used to manipulate streams and apply computational kernels expressed as shaders to streams. Connecting a shader program to a stream applies that program to all elements of the stream; combining streams concatenates the record definitions of those streams.

In conjunction with an optimizing compiler, these operators can manipulate shader programs in many useful ways, including specialization, without modifying the original source code. We demonstrate these operators in Sh, a metaprogramming shading language embedded in C++.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: real-time rendering, shader programming, graphics hardware

1 Introduction

Specialized high-level shading languages are a standard part of offline rendering systems and are now also available for real-time systems. Programs written in these languages are usually called shaders. Shaders can be downloaded to the Graphics Processing Units (GPUs) used in hardware graphics accelerators, to process attributes bound to streams of either vertices or pixel fragments. We can interpret a shader as a function that maps an input stream of homogeneous records to an output stream of homogeneous records. To allow strip-mined parallelism, operations on stream records execute independently.

Shading languages can be used to apply GPUs to non-traditional applications such as the solution of systems of equations or simulation. The results of these computations can be used for graphics applications, or the GPU can simply be used as a numerical engine.

As shaders become larger and more complex the ability to reuse and encapsulate code becomes more important. Shading languages can (and do) support standard forms of modularity such as subroutines, but other forms of modularity are possible. A dataflow model, for instance, is very natural for shaders, and visual dataflow languages are also often used in modeling and animation applications. Dataflow visual languages, in turn, are related to functional languages, where higher-order functions provide useful forms of modularity. Object-oriented constructions such as classes might also be useful in shading languages.

In this paper, we present an algebra over shader objects. This algebra consists of two binary operators: *connection* and *combination*. Using these operators, we can manipulate and specialize shaders without having to modify (or even have access to) the source code of the original shaders. The

addition of a shader algebra enables a functional style of programming: shaders become first-class objects that can be operated upon. The same operators can be used to express general-purpose stream processing.

Our algebra is demonstrated in the context of Sh, an embedded, metaprogrammable shading language implemented entirely within C++. Sh is similar to an operator overloaded matrix/vector library. When used in "immediate" mode, this is exactly what it is. However, sequences of operations on Sh types can be captured in a "retained" mode and compiled to another target, such as a GPU shading unit. The result is a language that is as expressive as a custom shading language but with all the power of C++ for creating modular shaders. For instance, it is easy to add new types to Sh, including operator overloading, or to use classes to build parameterized or customized versions of shaders, or to use function pointers to create procedurally parameterized shaders. In Sh, the shader algebra extends and augments these existing modularity capabilities.

2 Related Work

GPUs can be seen as a form of stream processor. Stream processors are distinguished from vector processors by the fact that on-chip temporary registers are available, so it is possible to perform a significant amount of arithmetic on each stream record before the result needs to be written back to memory. General-purpose stream processors have been developed and graphics systems have been implemented on them [Owens et al. 2000], but GPUs are also evolving towards a more general-purpose architecture similar to stream processors. General computations on GPUs have been studied by many researchers. In recent years many graphics, simulation, and numerical applications have been implemented on GPUs [Trendall and Stewart 2000; Hillesland et al. 2003; Bolz et al. 2003; Krüger and Westermann 2003].

A shading language is a domain-specific programming language for specifying shading computations. Shade trees [Cook 1984] captured expressions used to compute pixels so if the parameters of the lighting model changed in a raytracer, an image could be quickly recomputed without redoing intersection calculations. Peachey and Perlin [Peachey 1985; Perlin 1985] developed the idea of procedural textures and lighting, and also experimented with noise functions. Block shaders used a network of configurable modules [Abram and Whitted 1990] described using either a visual or a textual language. Explicit control was provided over the order of execution of modules so that side effects could be used for global communications. In their textual language, a netlist (rather than an algebra) was used to specify the connections between modules. However, the idea of connecting modules in a dataflow structure is closely related to our algebra. Dataflow languages (visual and otherwise) have also been used for procedural modeling [Hedelman 1984] and lighting networks [Slusallek et al. 1998]. The CONDOR system, for instance, was a constraint-based dataflow language [Kass 1992], which compiled a visual language to C++ code. The compiler was written in Lisp and Mathematica, and included support for symbolic derivatives and interval analysis. CONDOR was applied to both shading and geometric modeling, as well as numerical optimization — a suite of applications similar to the problems being tackled by modern GPUs. Operator-based systems such as GENMOD have also been developed for procedural geometry [Snyder and Kajiya 1992; Snyder 1992].

The RenderMan shading language has become a standard [Hanrahan and Lawson 1990; Upstill 1990; Apodaca and Gritz 2000; Pix 2000] and has strongly influenced other shading languages, particularly the idea of uniform and varying parameters. However, the RenderMan standard, although originally intended as a hardware API, is no longer used as such, and modern GPU shading languages need to target the hardware architecture of modern GPUs.

GPU shading languages include the OpenGL Shading Language [Kessenich et al. 2003; Rost 2004], the Stanford Real-Time Shading Language, [Proudfoot et al. 2001; Mark and Proudfoot 2001], Microsoft's HLSL, and Cg [Mark et al. 2003]. While useful for implementing shader kernels, these languages do not address the implementation of multipass algorithms involving several shaders, and binding of these language to the host application is loose. CgFX and the Direct3D Effects system provide mechanisms for specifying multipass combinations of shaders and GPU state for each pass, but provide only limited control mechanisms. The SGI Interactive Shading Language [Peercy et al. 2000] compiles shaders to a multipass implementation, but does not generate complex kernels, only primitive passes.

A system called Brook [Buck 2003; Buck et al. 2004] has been developed to target both stream processors and GPUs. Brook defines its own language as an extension of ANSI C. This is implemented as a preprocessor that maps Brook programs to a C++/Cg implementation. Brook's goals are similar to ours, but Brook currently supports a more extensive set of stream processing operators, such as reductions and scatter, as well as multidimensional streams. In the short term, we have focused instead on program specification, manipulation, and modularity abstractions.

We use a metaprogramming approach which gives us many modularity constructs for free. Metaprogramming is a useful mechanism for implementing domain-specific embedded languages and binding them to a host application [Elliott et al. 2000; Draves 1996]. Template metaprogramming [Veldhuizen 1999; Lee and Leone 1996; Dawes and Abrahams 2003] has become a popular method for reorganizing C++ code by using template rewriting rules as a functional language at compile time. However, template rewriting is inefficient, so it is hard to specify complex operations using it. Also, template metaprogramming only generates C++ code for the host, not for a GPU. Code generation (textual substitution) [Herrington 2003] is now a standard software engineering tool, and can be used to embed another language (such as an SQL query) into a host program and generate appropriate boilerplate binding code. Languages such as ruby, perl, tcl, or python can be used to specify textual transformations on code. However, the embedded code is still in a different language, integration with the host language may be incomplete unless a very sophisticated preprocessor is implemented, and the build process becomes more complex. Compiling such code now also depends on a tool which also has to be maintained (for instance, if the input is C++ plus some extensions, we need to maintain a parser that tracks the latest version of C++). Our implementation approach is closest to that of Tick CC [Poletto et al. 1999], which defined special types and operators for representing and combining program fragments. However, rather than extending C, we use the standard abstraction capabilities of C++ to define interfaces to our new operations. We then operate on our own internal representation of shaders to perform optimizations. Sh does *not* use template metaprogramming to implement its optimizer. Our approach makes it straightforward to implement (among other things) shader specialization [Guenter et al. 1995].

3 The Sh Library

Shader algebra operators have been implemented in Sh, an embedded metaprogrammed shading "language" [McCool et al. 2002]. Sh is available in open-source form; see `http://libsh.sourceforge.net`. Sh is actually a C++ library API rather than an independent language. It is based around a set of C++ types for representing small n-tuples and matrices upon which appropriate operators have been defined. In immediate mode, Sh operates like a standard graphics matrix/vector library, executing its operations on the host. However, sequences of Sh operations can also be "recorded" using a retained-mode mechanism similar to display lists and then compiled. Usually the Sh compiler targets a GPU, although we can also target other platforms, including other shading languages and the host CPU, via a modular backend system. Sh has its own optimizer that handles dead-code elimination and constant propagation (among other transformations). Sh also provides a runtime engine that transparently manages buffers and textures and controls the scheduling of multipass (virtualized) shaders.

Listing 1 gives a pair of Sh shader definitions for a (modified) Blinn-Phong lighting model [Blinn 1977]. First, we declare a set of named *parameters*: *phong_kd*, *phong_ks*, *phong_spec_exp*, *phong_light_position*, and *phong_light_color*. This pair of shaders also depends on the *modelview* and *perspective* matrix parameters, which are global declarations shared by many other shaders. Parameters are declared outside of a shader definition, and so can be acted upon in immediate mode on the host using whatever operators are defined for their types. Parameters are equivalent to `uniform` variables in RenderMan.

Then, we define a function `phong_init` that constructs two shaders, `phong_vert` and `phong_frag`. These shaders use the same types as we used for parameters, but with some of the declarations marked with `Input` and `Output` qualifiers. Actually, `ShColor3f` is a `typedef` for `ShColor<3,SH_TEMP,float>` and `ShInputColor3f` is a `typedef` for `ShColor<3,SH_INPUT,float>`, and similarly for the other types. Temporary values local to the shader can be defined with unqualified types or with `SH_TEMP` (the default second argument). Parameters and local temporaries are distinguished only by their location of definition: inside or outside a shader. Input and output *attributes* are bound to the input and output data channels of the shaders using a set of rules that depend on their order of declaration and their type. Input attributes act like `varying` variables. Note the distinction between *parameters* and *attributes*: this terminology is used extensively later.

Shaders can refer to parameters previously declared outside the shader definition. However, shaders cannot assign to parameters. When a shader is loaded into the appropriate GPU unit, copies of the values of the necessary parameters are loaded into constant registers (or bound to texture units; Sh considers textures to be a kind of array-valued parameter). If in immediate mode a new value is assigned to a parameter that is in use by a bound shader, the copy in the GPU is automatically updated. The net effect is that parameters act like external variables relative to the definition of the shader, and C++ scope rules can be used to manage the binding of parameters to shaders.

Since parameters are bound to shaders using the scope rules of C++ and due to the way we "capture" a sequence of Sh operations, all C++ modularity and control constructs can be used to organize Sh shaders. Listing 2 gives a more involved Sh example that uses a class to control access to the parameters, uses a template argument to specify the number of light sources, uses a construction argument to pass in the resolution of the texture maps, uses a template `struct` and the alternative template-based declaration of Sh types to coordinate the values passed between vertex and fragment shaders, and uses C++ control constructs to loop over the light sources. We can also use C++ functions and function pointers (the latter can be used to simulate the `interface` mechanism of Cg, among other applications). Users can also define their own types and add them to the system, including operator overloading. For instance, we have developed types to represent compressed textures, sparse matrices, complex numbers, quaternions, and geometric algebra constructs.

```
ShMatrix4x4f modelview;          // MCS to VCS transformation
ShMatrix4x4f perspective;        // VCS to DCS transformation
ShColor3f phong_kd;              // diffuse color
ShColor3f phong_ks;              // specular color
ShAttrib1f phong_spec_exp;       // phong specular exponent
ShPoint3f phong_light_position; // VCS light position
ShColor3f phong_light_color;     // light source color
ShProgram phong_vert, phong_frag;
void phong_init () {
  phong_vert = SH_BEGIN_PROGRAM("gpu:vertex") {
    ShInputNormal3f nm;        // IN(0): normal vector (MCS)
    ShInputPosition3f pm;      // IN(1): position (MCS)
    ShOutputNormal3f nv;       // OUT(0): normal (VCS)
    ShOutputVector3f lv;       // OUT(1): light-vector (VCS)
    ShOutputVector3f vv;       // OUT(2): view vector (VCS)
    ShOutputColor3f ec;        // OUT(3): irradiance
    ShOutputPosition4f pd;     // OUT(4): position (HDCS)
    ShPoint4f pvt = modelview | pm;
    vv = -pvt(0,1,2);
    lv = normalize(phong_light_position + vv);
    nv = normalize(modelview | nm);
    ec = phong_light_color * pos(nv | lv);
    pd = perspective | pvt;
  } SH_END;
  phong_frag = SH_BEGIN_PROGRAM("gpu:fragment") {
    ShInputNormal3f nv;          // IN(0): normal (VCS)
    ShInputVector3f lv;          // IN(1): light-vector (VCS)
    ShInputVector3f vv;          // IN(2): view vector (VCS)
    ShInputColor3f ec;           // IN(3): irradiance
    ShOutputColor3f fc;          // OUT(0): fragment color
    vv = normalize(vv);
    lv = normalize(lv);
    nv = normalize(nv);
    ShVector3f hv = normalize(lv + vv);
    fc = phong_kd * ec +
         phong_ks * pow(pos(hv | nv), phong_spec_exp);
  } SH_END;
}
```

Listing 1: Blinn-Phong lighting model shaders.

In addition to the ability to use C++ modularity and control constructs, the Sh metaprogramming approach has other advantages. First, shaders can be constructed algorithmically. For instance, we have generated shaders to represent arbitrary polygons procedurally by finding a minimal CSG tree and generating code to evaluate it. Secondly, no glue code is required to bind shaders to the host application; they act like part of it. Third, Sh shaders can be debugged using a standard IDE by first running them in immediate mode, where the code executes on the host (although we are developing more specialized debugging tools).

```
template <int NLIGHTS>
class BlinnPhong {
  public:
    ShTexture2D<ShColor3f> kd;
    ShTexture2D<ShColor3f> ks;
    ShAttrib1f spec_exp;
    ShPoint3f light_position[NLIGHTS];
    ShColor3f light_color[NLIGHTS];
    template <ShVariableKind IO> struct VertFrag {
      ShPoint<4,IO,float> pv;             // position (VCS)
      ShTexCoord<2,IO,float> u;           // texture coordinate
      ShNormal<3,IO,float> nv;            // normal (VCS)
      ShColor<3,IO,float> ec;             // total irradiance
    };
    ShProgram vert, frag;
    BlinnPhong (int res) : kd(res,res), ks(res,res) {
      vert = SH_BEGIN_PROGRAM("gpu:vertex") {
        ShInputNormal3f nm;       // normal vector (MCS)
        ShInputTexCoord2f u;      // texture coordinate
        ShInputPosition3f pm;     // position (MCS)
        VertFrag<SH_OUTPUT> vf;
        ShOutputPosition4f pd;    // position (HDCS)
        vf.pv = modelview | pm;
        vf.u = u;
        vf.nv = normalize(modelview | nm );
        pd = perspective | vf.pv;
        for (int i=0; i<NLIGHTS; i++) {
          ShVector3f lv =
            normalize(light_position[i] - vf.pv(0,1,2));
          vf.ec += light_color[i] * pos(vf.nv|lv);
        }
      } SH_END;
      frag = SH_BEGIN_PROGRAM("gpu:fragment") {
        VertFrag<SH_INPUT> vf;
        ShOutputColor3f fc;          // fragment color
        ShVector3f vv = normalize(-vf.pv(0,1,2));
        ShNormal3f nv = normalize(vf.nv);
        fc = kd(vf.u) * vf.ec;
        ShColor3f kst = ks(vf.u);
        for (int i=0; i<NLIGHTS; i++) {
          ShVector3f lv =
            normalize(light_position[i] - vf.pv(0,1,2));
          ShVector3f hv = normalize(lv + vv);
          fc += kst * pow(pos(hv|nv),spec_exp) * light_color[i];
        }
      } SH_END;
    }
};
```

Listing 2: Encapsulated Blinn-Phong lighting model.

4 Operators

Two operators are defined: connection, which is defined as functional composition or application, and combination, which is equivalent in the case of Sh to concatenation of source code. These operators are defined over shader objects and create new shader objects. Shader objects can be thought of as functions that take an ordered sequence of n inputs and map them to an ordered sequence of m outputs.

4.1 Connection

Suppose we have a shader object `q1` with n inputs and k outputs and another shader object `p1` with k inputs and m outputs. The *connection* operator creates a new shader object with n inputs and m outputs by taking the outputs of `q1` and feeding them in the same order to the inputs of `p1`. In other words, it performs functional composition.

We denote this operator in Sh using the "`<<`" operator, with inputs on the right and outputs on the left. For instance, the k outputs of `q1` can connected to the k inputs of `p1` using "`p1 << q1`".

The outputs of `q1` must match the inputs of `p1` in number, size, and type (both storage and semantic). These are checked dynamically, at C++ runtime.

4.2 Combination

Suppose we are given two shader objects `p2` and `q2`. Let `p2` have n inputs and m outputs, and let `q2` have k inputs and ℓ outputs. We define the *combination* of `p2` and `q2` to have $n+k$ inputs and $m+\ell$ outputs, with the inputs and outputs of `p2` appearing first, followed by the inputs and outputs of `q2`. The computations of `p2` and `q2` are both performed, with the local variables of each in different scopes.

We denote this operator in Sh using "`&`", and so the combination of `p2` and `q2` can be denoted with "`p2 & q2`". Note that "`&`" binds more loosely than "`<<`".

Because of the way Sh is defined, the combination operator is equivalent to the concatenation of the source code of the input shaders, using two separate scopes. Such a concatenation would ensure that the inputs and outputs of `p2` are declared before `q2`, and so would give the same result as defined above.

Both operators do not simply build a network of shaders. They actually operate on the internal representation of the shader to build a completely new shader, which is ultimately run through the full suite of optimizations and virtualizations supported by the Sh backend.

For vertex and fragment shaders, a special `ShPosition*` semantic type is defined which is semantically equivalent to an `ShPoint*` but binds to the special position input and output of these shader units on GPUs. The last definition always dominates, so if a position is computed in two shaders that are combined, only the position in the second shader will be used, and the first position will be converted to a point.

5 Manipulators

Use of shader algebra operators alone can result in redundant computation. However, the "`<<`" operator, in conjunction with the optimizer in the Sh compiler (particularly dead code removal) and the definition of some simple "glue" shader programs, can be used to specialize shaders and eliminate such redundant computations.

For instance, suppose we combine two shader programs and the resulting shader computes the same value twice (in two different ways, so we cannot discover this fact automatically). We can define a simple program that copies its inputs to its outputs *except* for one of the redundant results. This simple shader can be connected to the output of the combined shader and the Sh dead code eliminator will remove the redundant computation.

Unfortunately, to satisfy the type rules for connecting shader programs, we need to define the interface of each such glue shader to match the particular interface types of the given base shader. This is annoying if all we want to do is rearrange the inputs and outputs of a shader. We would like to be able to specify simple rearrangements inline, within an expression.

Sh provides some shortcuts, similar to manipulators in the C++ `iostream` library, for manipulating the input and output channels of shaders. These manipulators are really

functions that return instances of either shader program objects or instances of special manipulator classes. Manipulator classes store information about the particular manipulation required. When combined with a shader in an expression, the appropriate glue shader is automatically generated, using introspection over the shader programs it is combined with, to perform the desired manipulation. This second approach is used to automatically resolve type issues.

There are three kinds of manipulators: primitive manipulators that build primitive shaders directly (returning ShProgram objects rather than manipulator objects), those that take a fixed number of inputs, and those that consume all inputs. Primitive and fixed-input manipulators can be combined with the "&" operator to create more complex manipulations. Expandable manipulators that consume all available inputs can only be combined with programs and other manipulators using the "<<" operator. Supported primitive manipulators include the following:

`keep<T>(int n = 1)`: Generates a program that copies `n` channels of type `T` from its input to its output.

`lose<T>(int n = 1)`: Generates a program that reads `n` channels of type `T` from its input and discards them (no outputs).

`access(A t)`: Inserts a texture lookup into a channel. The input is a texture coordinate of the appropriate dimensionality (depending on type `A`) and the output is the return/storage type of the texture type `A`.

Supported fixed-input manipulators include the following:

`shDup(int n = 2)`: Reads one input channel, and creates `n` duplicates on its output.

`shKeep(int n = 1)`: Copies `n` channels from its input to its output.

`shLose(int n = 1)`: Reads `n` channels from its input and discards them (no outputs).

Expandable input manipulators include the following:

`shExtract(T i)`: Moves the referenced channel to the beginning of the attribute list, rearranging the other channels to close the gap.

`shDrop(T i)`: Discards the referenced channel, and rearranges the other channels to close the gap.

`shSwizzle(T i0, T i1, ...)`: Performs a swizzle of the given indices. The inputs are rearranged into the order given by the arguments. Note that duplication and deletion of inputs is also possible.

`shRange(T i0)(T i1, T i2)`: Takes an arbitrary sequence of `(T i)` (to identify a single channel) or `(T i1, T i2)` (to identify a channel range) postfixes. This manipulator is an alternative form for specifying a swizzle.

To identify channels by position an integer is used for `T`. Negative numbers may be used to specify the position of a channel counted from the end of the attribute list, with -1 identifying the last channel. Channels can also be identified by name, using `const char *` for `T`. Names are unfortunately not automatically assigned to Sh variables, since C++ has no standard way to find out the names of its own variables. A `name` method is therefore provided on all Sh types to provide string names for such identification purposes (other metadata can also be attached for introspection purposes).

Combinations of `shKeep` and `shLose` with "&" can be used to describe mappings that retain a subset of outputs. The `shExtract` manipulator is useful when combined with currying, described later, to replace a named attribute with a parameter. The `shSwizzle` (not to be confused with the swizzle operator on tuples) and `shRange` manipulators are generally useful when adapting the interface of some shaders to others, or to the order in which data is presented.

These manipulators cannot handle all possible cases. In particular, types are retained, and sometimes extra computation (such as normalization of vectors) is required. Manipulators are just a convenience; more complex adaptation of the input and output of shaders, including type casts and any additional computation required, can always be accomplished by defining suitable "glue" shaders.

6 Streams

Stream objects are represented in Sh using the ShChannel template class and ShStream class. Extensions of the shader algebra to streams enable a general-purpose stream processing computational model.

A channel is a sequence of elements of the type given as its template argument. Channels are an abstraction and the channel data representation is opaque, but channel data can be used as a vertex array input. The Sh runtime uses the most efficient operations and representations available for managing buffers, and timestamps updates to data to avoid data transfers whenever possible. For instance, under OpenGL the NVIDIA backend uses textures and pbuffers, while the ATI backend uses überbuffers. When/if superbuffers are available, Sh will use those, or whatever buffer management interface is most efficient on a given platform.

Streams are containers for several channels of data, and are specified by combining channels (or other streams) with the "&" operator. Streams only *refer* to channels, they do not create copies. Streams may not refer to themselves as components. A channel can still be referenced as a separate object, and can also be referenced by more than one stream at once. For convenience, an ShChannel of any type can also be used directly as a single-channel stream.

Sh uses a reference-counting garbage collection scheme. Most Sh types are in fact smart pointers to separate data items. Even if a channel is destroyed (explicitly or implicitly), if a stream refers to this data the memory will not be released. Type information is also represented internally in most Sh objects to support dynamic type checking.

In addition to being viewed as a sequence of channels, a stream can also be seen as a sequence of homogeneous records, each record being a sequence of elements from each component channel. Stream programs or *kernels* conceptually map an input record type to an output record type. If an ShProgram is compiled with the `"gpu:stream"` or `"cpu:stream"` profile, it can be applied to streams. Stream kernels are applied in parallel (conceptually) to all records in the stream.

The *connect* operator is overloaded to permit the application of kernels to streams. For instance, a program `p` can be applied to an input stream `a` and its output directed to an output stream `b` as follows:

```
b = p << a;
```

When specified, the above stream operation will execute immediately, and will return when it is complete (later on we

plan to add a retained mode to permit greater optimization). At the point of execution, Sh will check (dynamically) that the input and output types of the program match the types of the input and output streams.

Use of "`p << a`" alone creates an unevaluated program kernel, which is given the type `ShProgram` (and can be assigned to a variable of this type, if the user does not want execution to happen immediately). What actually happens is that input attributes are replaced with fetch operators in the intermediate language representation of the program. These fetch operators are initialized to refer to the given stream's channels. Such bound program objects can also be interpreted as procedural streams. A bound but unevaluated program will only be executed when it is assigned to an output stream.

The implementation of the `<<` operator permits currying. If a shader is applied to a stream with an insufficient number of channels, an unevaluated program with fewer inputs is returned. This program requires the remainder of its inputs before it can execute.

In a functional language, currying is usually implemented with deferred execution. Since in a pure functional language values in variables cannot be changed after they are set, this is equivalent to using the value in effect at the point of the curry. However, in an imperative language, we are free to modify the value provided to the curried expression. We *could* copy the value at the point of the curry, but this would be expensive for stream data. Instead, we use deferred read semantics: later execution of the program will use the value of the stream in effect at the of point actual execution, *not* the value in effect at the point of the currying. This is useful in practice, as we can create (and optimize) a network of kernels and streams in advance and then execute them iteratively.

The `<<` operator can also be used to apply programs to Sh tuples. A mixture of tuple and stream inputs may be used. In this case, the tuple is interpreted as a stream all of whose elements are the same value. The same by-reference semantics are applied for consistency. Effectively, an input "varying" attribute is converted into a "uniform" parameter, a useful operation.

Since we provide an operator for turning a varying attribute into a uniform parameter, we also provide an inverse operator for turning a parameter into an attribute. Given program `p` and parameter `x`, the following removes the dependence of `p` on `x`, creating a new program object `q`:

```
ShProgram q = p >> x;
```

The parameter is replaced by a new attribute of the same type, pushed onto the *end* of the input attribute list.

The "`&`" operator can be applied to streams, channels, or tuples on the left hand side of an assignment. This can be used to split apart the output of a kernel. For instance, let `a`, `b`, and `c` be channels or streams, and let `x`, `y`, and `z` be streams, channels, or tuples. Then the following binds a program `p` to some inputs, executes it, and extracts the individual channels of the output:

```
(a & b & c) = p << x << y << z;
```

This syntax also permits Sh programs to be used as subroutines (let all of `a`, `b`, `c`, `x`, `y`, and `z` be tuples).

7 Examples

Listing 3 gives an example of the use of shader algebra operators to manipulate shaders. This example takes the Blinn-Phong fragment shader given in Listing 1 and converts the parameters *phong_kd* and *phong_ks* into attributes, and then feeds these new attributes with the results of texture reads. The coordinates fed into these texture reads, in turn, are generated by a noise function, generating the image shown in Figure 3 [Hart et al. 1999]. Note that the light and dark wood actually have different reflectances. With some changes in parameters this shader can also be used for marble. One advantage of Sh that should be pointed out here is that the noise function is opaque, but actually may be implemented using hidden texture reads. Sh manages both textures and shaders in an integrated manner, something not done by other GPU shading languages, and made possible by the close binding to the host application. A diagrammatic representation of the effect of the various manipulators and shader algebra operators in this example is shown in Figure 2.

```
ShAttrib2f wood_freq;
ShAttrib1f wood_scale;
ShAttrib1f wood_noise_scale;
ShTexture1D<ShColor3f> wood_kd(256);
ShTexture1D<ShColor3f> wood_ks(256);
ShProgram wood_vert, wood_frag;

void wood_init() {
  // modify phong_vert to pass through model space position
  wood_vert = phong_vert & keep<ShPoint3f>();
  wood_vert = wood_vert << (shKeep(1) & shDup(2));

  // define kernel to generate texture coords from noise
  ShProgram wood_frag_inc = SH_BEGIN_PROGRAM("gpu:fragment") {
    ShInputPoint3f x;       // IN(0): model space position
    ShOutputTexCoord1f u;   // OUT(1): texture coordinate
    ShPoint3f scaledX = x(0,1,2)*wood_scale*0.5;
    ShAttrib1f noise = sturbulence(scaledX);
    u = frac(wood_scale*(x(1,2)|x(1,2))+wood_noise_scale*noise);
  } SH_END;

  // make two copies of texture coords (one for ks, one for kd)
  wood_frag = shDup(2) << wood_frag_inc;
  // feed texture coords into texture lookups for ks and kd
  wood_frag = (access(wood_kd) & access(wood_ks)) << wood_frag;
  // convert ks and kd parameters to input attributes
  ShProgram wood_frag_base = phong_frag >> phong_kd >> phong_ks;
  // replace new inputs with our noise-based texture lookups.
  wood_frag = wood_frag_base << (shKeep(4) & wood_frag);
}
```

Listing 3: Wood defined by modifying texture coordinates of Blinn-Phong lighting model.

Listing 4 gives a stream program to update the state of a particle system [Sims 1990]. This program implements simple Newtonian physics and can handle collisions with both planes and spheres. The particles are rendered by feeding the particles back through the GPU as a vertex array (code for this is not shown). Screenshots are shown in Figure 4. This example demonstrates the use of deferred read semantics for currying. The `state` stream is defined and bound to the `particle` program object along with some uniform parameters. The result is assigned to the `update` program object, which triggers compilation and optimization. The `update` object now has compiled-in access to the channels

pointed to by the `state` stream. The inner loop is very simple and fast. In particular, all shader compilation is done during setup. This example demonstrates that the use of shader algebra operations in conjunction with the stream and channel objects permits the application of Sh to non-shading (general-purpose) applications such as simulation.

Shader algebra operations can be used for many other things. For instance, they can be used to combine "light" and "surface" shaders, or to "plug in" post-processors such as tone mapping or halftoning. Any parameter or attribute can also be replaced with a texture. In fact, it is just as easy to replace any attribute or parameter with a *procedure*, or to perturb attributes like normals to implement bump mapping on any surface shader. All these possibilities are demonstrated in Figures 1 and 5. These images were all generated by combining a small number of basic shaders with simple expressions.

Specialization is also very useful. For instance, we have implemented a "universal" vertex shader that does a number of standard operations, including transformation of normals, tangents, vertices, and texture coordinates. We then just use shader algebra expressions to specialize it to generate only the outputs we need for a given fragment shader. Only in exceptional circumstances do we have to write vertex shader code. In this case named outputs and inputs and the named versions of the manipulators are especially useful.

A shader algebra expression is not always the best way to manipulate a shader. As mentioned earlier, subroutines, classes, and other mechanisms are also useful to modularize shaders. However, shader algebra expressions and manipulators can express simple and common operations on shaders in a relatively straightforward and expressive manner.

Figure 3: Images generated with the simple Blinn-Phong shader, two parameter settings for the encapsulated Blinn-Phong shader, and the wood shader derived using shader algebra operators. See Listings 1, 2, and 3.

```
// SETUP
particle = SH_BEGIN_PROGRAM("gpu:stream") {
  ShInOutPoint3f Ph, Pt;
  ShInOutVector3f V; ShInputVector3f A;
  ShInputAttrib1f delta;
  Pt = Ph; // Physical state update
  A = cond(abs(Ph(1)) < 0.05, ShVector3f(0.,0.,0.), A);
  V += A * delta;
  V = cond((V|V) < 1.0, ShVector3f(0.0, 0.0, 0.0), V);
  Ph += (V + 0.5 * A) * delta;
  struct { // Declare spheres
    ShInputPoint3f center;
    ShInputAttrib1f radius;
  } spheres[num_spheres];
  ShAttrib1f mu(0.1), eps(0.3);
  for (int i = 0; i < num_spheres; i++) { // Sphere collisions
    ShPoint3f C = spheres[i].center;
    ShAttrib1f r = spheres[i].radius;
    ShVector3f PhC = Ph - C;
    ShVector3f N = normalize(PhC);
    ShPoint3f S = C + N * r;
    ShAttrib1f collide = ((PhC|PhC) < r * r) * ((V|N) < 0);
    Ph = cond(collide, Ph - 2.0 * ((Ph - S)|N) * N, Ph);
    ShVector3f Vn = (V|N) * N;
    ShVector3f Vt = V - Vn;
    V = cond(collide, (1.0 - mu) * Vt - eps * Vn, V);
  }
  ShAttrib1f under = Ph(1) < 0.0; // Collide with ground
  Ph = cond(under, Ph * ShAttrib3f(1.0, 0.0, 1.0), Ph);
  ShVector3f Vn = V * ShAttrib3f(0.0, 1.0, 0.0);
  ShVector3f Vt = V - Vn;
  V = cond(under, (1.0 - mu) * Vt - eps * Vn, V);
  Ph(1) = cond(min(under, (V|V) < 0.1), ShPoint1f(0.0f), Ph(1));
  ShVector3f dt = Pt - Ph; // Avoid lines disappearing
  Pt = cond((dt|dt) < 0.02, Pt + ShVector3f(0.0, 0.02, 0.0), Pt);
} SH_END;

// define stream specifying current state
ShStream state = (pos & pos_tail & vel);
// define update operator, bind to inputs
ShProgram update = particle << state << gravity << delta;
// add collision specifications (compiler uses uniforms)
for (int i = 0; i < num_spheres; i++) {
  update = update << ShAttrib3f(scene_spheres[i].center[0],
                                scene_spheres[i].center[1],
                                scene_spheres[i].center[2])
                  << ShAttrib1f(scene_spheres[i].radius);
}
...
// IN INNER LOOP
// execute state update (input to update is compiled in)
state = update;
```

Listing 4: Particle system state update implemented as a stream program.

8 Conclusions

We have presented an algebraic approach to modularity in shaders. This approach is based on a treatment of shaders and stream kernels as objects that can be acted upon. Extension of the operators in this algebra to the application of programs to streams and tuples permit the invocation of stream computations and the reorganization of attributes and parameters. In addition to modularity, the shader algebra operations also enable a general-purpose stream processing computational model.

Several issues remain in the Sh implementation. Arbitrary length shaders can be defined in Sh but may not execute on a given GPU target. We currently virtualize some resources (such as the number of outputs) but not all. This adversely affects portability. We are in the process of implementing virtualization algorithms to convert long GPU programs into multipass implementations automatically. Multipass implementation can also be used to support data-dependent control constructs. Sh supports syntax for data-dependent control constructs but currently these can only be used for CPU compilation targets. However, in the meantime C++ con-

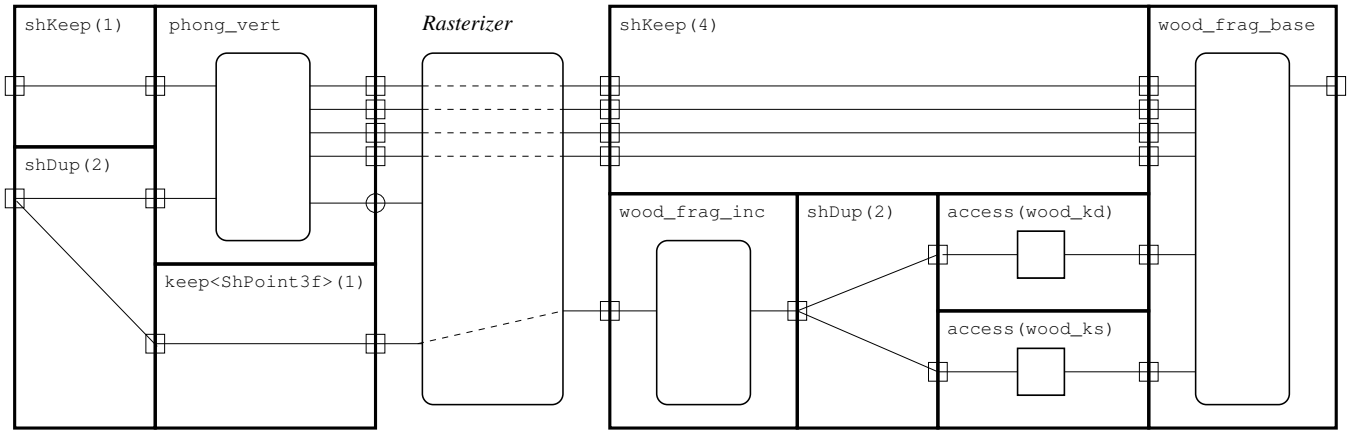

Figure 2: Diagrammatic representation of manipulator expressions to compute wood shaders from phong shaders.

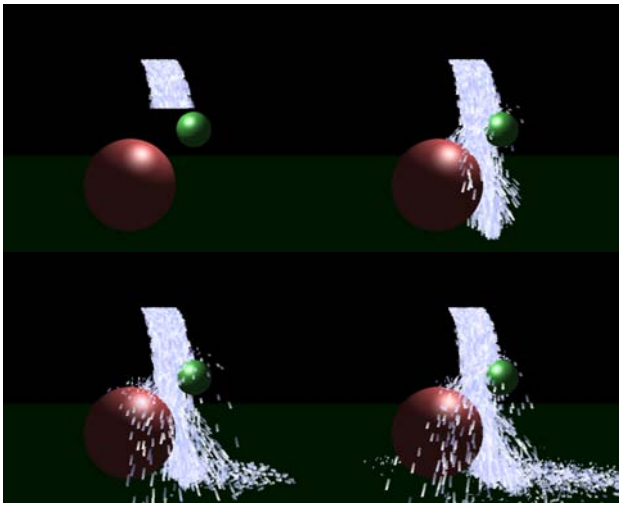

Figure 4: Frames from the particle system animation corresponding to Listing 4.

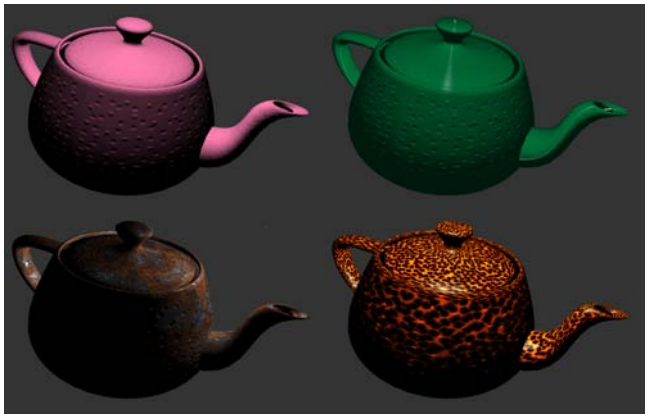

Figure 5: The normals (and/or tangents) used for any lighting model can be perturbed to implement bump mapping.

trol constructs can be used for conditional compilation and unrolled loops.

The examples given here for the shader algebra are limited in complexity. Once virtualization is supported, modularity will become much more important. For instance, a ray tracer framework might use shader algebra operations to "plug in" shaders for surfaces, after which the overall system (including the raytracer core as well as the plugged-in shader code) could then be compiled and optimized as a unit.

The shader algebra could be extended in various ways. Stride, sorting, scatter/gather, and indexing operations could be added to manipulate streams. Lifting arithmetic operators to programs and streams would also be useful. Applying addition to two program objects should generate a simple program and connect it to the input programs to add all output channels together. Similarly, operators like "+=" could be overloaded on streams to support the invocation of reduction operators. These additions would lead to a higher-level stream processing language that manipulates streams in the same way that shaders manipulate records.

This stream language would operate in "immediate mode", but it would also be useful to define a "retained mode" for sequences of stream operations (in the same manner as shaders retain sequences of tuple operations) so that they could be precompiled and optimized.

Acknowledgements

This research was sponsored by grants from ATI, Communications and Information Technology Ontario (now a division of the Ontario Centres of Excellence), the National Science and Engineering Research Council of Canada, Bell University Labs, the Canadian Foundation for Innovation, the Ontario Research and Development Challenges Fund and by hardware donations from ATI and NVIDIA. The cooperation and assistance of other members of the Computer Graphics Lab is also appreciated, especially Zheng Qin, Jack Wang, Zaid Mian, Ju-Lian Kwan, Edwin Vane, Filip Spaček, David Larsson and Gabriel Moreno-Fortuny.

References

ABRAM, G. D., AND WHITTED, T. 1990. Building block shaders. *Computer Graphics (Proc. SIGGRAPH) 24*, 4 (Aug.), 283–288.

APODACA, A. A., AND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

BLINN, J. F. 1977. Models of light reflection for computer synthesized pictures. *Computer Graphics (Proc. SIGGRAPH)* (July), 192–198.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. on Graphics (Proc. SIGGRAPH) 22*, 3 (July), 917–924.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics (Proc. SIGGRAPH) 23*, 3 (August).

BUCK, I. 2003. BrookGPU web site. *http://graphics.stanford.edu/projects/brookgpu/*.

COOK, R. L. 1984. Shade trees. *Computer Graphics (Proc. SIGGRAPH) 18*, 3 (July), 223–231.

DAWES, B., AND ABRAHAMS, D. 2003. Boost++ web site. *http://www.boost.org*.

DRAVES, S. 1996. Compiler generation for interactive graphics using intermediate code. In *Dagstuhl Seminar on Partial Evaluation*, 95–114.

ELLIOTT, C., FINNE, S., AND DE MOOR, O. 2000. Compiling embedded languages. In *SAIG/PLI*, 9–27.

GUENTER, B., KNOBLOCK, T., AND RUF, E. 1995. Specializing shaders. In *Proc. SIGGRAPH*, 343–350.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, 289–298.

HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. 1999. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, ACM Press, 45–53.

HEDELMAN, H. 1984. A data flow approach to procedural modeling. *IEEE CG&A 3*, 1 (January), 16–26.

HERRINGTON, J. 2003. *Code Generation in Action*. Manning Publications.

HILLESLAND, K. E., MOLINOV, S., AND GRZESZCZUK, R. 2003. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Trans. on Graphics (Proc. SIGGRAPH) 22*, 3 (July), 925–934.

KASS, M. 1992. CONDOR: Constraint-based dataflow. In *Proc. SIGGRAPH*, 321–330.

KESSENICH, J., BALDWIN, D., AND ROST, R. 2003. *OpenGL 2.0 Shading Language*, 1.051 ed., Feb.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. on Graphics (Proc. SIGGRAPH) 22*, 3 (July), 908–916.

LEE, P., AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, 137–148.

MARK, W. R., AND PROUDFOOT, K. 2001. Compiling to a VLIW fragment pipeline. In *Proc. Graphics Hardware*, SIGGRAPH/Eurographics.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. on Graphics (Proc. SIGGRAPH) 22*, 3 (July), 896–907.

MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Proc. Graphics Hardware*, 57–68.

OWENS, J. D., DALLY, W. J., KAPASI, U. J., RIXNER, S., MATTSON, P., AND MOWERY, B. 2000. Polygon rendering on a stream architecture. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 23–32.

PEACHEY, D. 1985. Solid texturing of complex surfaces. *Computer Graphics (Proc. SIGGRAPH) 19*, 3 (July), 279–286.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proc. SIGGRAPH*, 425–432.

PERLIN, K. 1985. An image synthesizer. *Computer Graphics (Proc. SIGGRAPH) 19*, 3 (July), 287–296.

PIXAR. 2000. *The RenderMan Interface, version 3.2*, July.

POLETTO, M., HSIEH, W. C., ENGLER, D. R., AND KAASHOEK, M. F. 1999. 'C and tcc: a language and compiler for dynamic code generation. *ACM Trans. on Programming Languages and Systems 21*, 2, 324–369.

PROUDFOOT, K., MARK, W. R., HANRAHAN, P., AND TZVETKOV, S. 2001. A real-time procedural shading system for programmable graphics hardware. *Computer Graphics (Proc. SIGGRAPH)* (Aug.), 159–170.

ROST, R. J. 2004. *OpenGL Shading Language*. Addison-Wesley.

SIMS, K. 1990. Particle animation and rendering using data parallel computation. *Computer Graphics (Proc. SIGGRAPH) 24*, 4 (August), 405–413.

SLUSALLEK, P., STAMMINGER, M., HEIDRICH, W., POPP, J.-C., AND SEIDEL, H.-P. 1998. Composite lighting simulations with lighting networks. *IEEE CG&A 18*, 2 (Mar.), 22–31.

SNYDER, J. M., AND KAJIYA, J. T. 1992. Generative modeling: A symbolic system for geometric modeling. *Computer Graphics (Proc. SIGGRAPH) 26*, 2 (July), 369–378.

SNYDER, J. M. 1992. Interval analysis for computer graphics. *Computer Graphics (Proc. SIGGRAPH) 26*, 2 (July), 121–130.

TRENDALL, C., AND STEWART, A. J. 2000. General calculations using graphics hardware, with applications to interactive caustics. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)*, Springer, 287–298.

UPSTILL, S. 1990. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley.

VELDHUIZEN, T. L. 1999. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*.